Hiring Assignment: The Smart Meeting Scheduler

Introduction

Welcome! This assignment is designed to assess your problem-solving, software design, and programming skills in a scenario that mirrors real-world challenges. The goal is to build a backend service that intelligently schedules meetings for a group of users.

We expect this project to be treated like a professional piece of work. We're interested in your approach to the problem, the quality of your code, and your design decisions.

The Problem: Scheduling is Hard

Finding a time that works for everyone in a team is a common and often frustrating problem. A simple scheduler might find the *first* available slot, but a *smart* scheduler finds the *best* one.

Your task is to build a **Smart Meeting Scheduler API**. This service will accept a request to schedule a new meeting for a list of participants and will find the optimal time slot based on their existing calendars and a set of scheduling preferences.

Core Requirements (Must-Haves)

You must build a backend service that exposes a RESTful API. You are free to choose the language, framework, and database you are most comfortable with (e.g., Node.js/Express, Python/Flask, Go, Java/Spring, etc.).

Data Requirements

Your application will need to manage data for users and their calendar events. It's up to you to design the specific database schema or data models, but they must be able to store the following information:

- **For each user:** A unique identifier and their name.
- **For each calendar event:** A title for the event, a specific start time, a specific end time, and a clear association with the user to whom it belongs.

API Endpoints

1. **POST /schedule**
    - This is the core endpoint of the application.
    - **Request Body**:
      {

```
  "participantIds": ["user1", "user2", "user3"],
  "durationMinutes": 60,
  "timeRange": {
    "start": "2024-09-01T09:00:00Z",
    "end": "2024-09-05T17:00:00Z"
  }
}
```

- **Logic**:
  1. For the given participantIds, fetch all their existing calendar events.
  2. Within the specified timeRange, identify all common time slots where every participant is available for the required durationMinutes.
  3. From these available slots, select the **optimal** one based on the scoring criteria defined in the "Smart Challenge" section.
  4. "Book" the meeting by creating new calendar events for each participant for the chosen time slot.
- **Success Response (201 Created)**:

```
{
  "meetingId": "meeting-xyz-123",
  "title": "New Meeting",
  "participantIds": ["user1", "user2", "user3"],
  "startTime": "2024-09-02T10:00:00Z",
  "endTime": "2024-09-02T11:00:00Z"
}
```

- **Error Response (409 Conflict)**: If no common slot can be found.

```
{
  "error": "No available time slot found for all participants."
}
```

2. **GET /users/:userId/calendar**
   - Returns all calendar events for a given user within a specified time window.
   - **Query Parameters**: ?start=...&end=... (ISO 8601 format)
   - **Success Response (200 OK)**:

```
[
  {
    "title": "Existing Meeting",
    "startTime": "2024-09-02T14:00:00Z",
    "endTime": "2024-09-02T15:00:00Z"
```

```
  }
]
```

Initial Data

For testing purposes, your service should start with some pre-populated calendar data for a few users. You can hard-code this, load it from a file, or use a database seeding script.

The "Smart" Challenge (Logical Thinking)

Simply finding the first available slot is not enough. Your service must implement a scoring algorithm to determine the **best** slot. This is the core problem-solving aspect of the assignment.

You should define a function that scores each potential slot based on a set of heuristics. Here are some ideas, but **we encourage you to add your own**:

- **Prefer Earlier Slots**: A meeting at 10 AM might be better than one at 4 PM.
- **Minimize Calendar Gaps**: Avoid creating awkward 30-minute gaps between meetings. A slot that is back-to-back with another meeting might be preferable to one that leaves a small, unusable window of time.
- **Respect "Buffer" Time**: Penalize slots that are immediately adjacent to other meetings. A 15-minute buffer is often desirable.
- **Working Hours**: Prefer slots that fall within standard working hours (e.g., 9 AM - 5 PM).

The slot with the "best" score (e.g., lowest penalty or highest score) should be chosen. Please document your chosen heuristics and scoring logic in the README.md.

What to Submit

1. **A link to a Git repository** (e.g., on GitHub, GitLab) containing your complete source code.
2. A **README.md** file that includes:
    - A clear explanation of your design and architectural choices, including your data models.
    - A detailed description of your algorithm for finding the optimal meeting time and the heuristics you implemented.
    - Step-by-step instructions on how to build, configure, and run the application locally.

    - Instructions on how to run any tests you've written.
3. (Bonus) A Postman collection or similar artifact to make testing your API endpoints easier.

Evaluation Criteria

We will be evaluating your submission based on the following:

- **Correctness**: Does the application meet all the core requirements?
- **Problem-Solving**: How effectively did you design and implement the slot-finding and scoring algorithm?
- **Code Quality**: Is the code clean, well-structured, readable, and maintainable?
- **Data Modeling**: Is the database schema well-designed for the application's needs?
- **API Design**: Is the API intuitive, consistent, and well-designed?
- **Testing**: Have you included meaningful unit or integration tests to ensure your logic is correct?
- **Documentation**: How clear is your README.md?

Good luck! We look forward to seeing your solution.